Distributed, Secure Load Balancing with Skew, Heterogeneity, and Churn

Jonathan Ledlie and Margo Seltzer
Division of Engineering and Applied Science
Harvard University

***Abstract*— Numerous proposals exist for load balancing in peer-to-peer (p2p) networks. Some focus on namespace balancing, making the distance between nodes as uniform as possible. This technique works well under ideal conditions, but not under those found empirically. Instead, researchers have found heavy-tailed query distributions (skew), high rates of node join and leave (churn), and wide variation in node network and storage capacity (heterogeneity). Other approaches tackle these less-than-ideal conditions, but give up on important security properties. We propose an algorithm that both facilitates good performance and does not dilute security. Our algorithm, *k-Choices*, achieves load balance by greedily matching nodes' target workloads with actual applied workloads through limited sampling, and limits any fundamental decrease in security by basing each nodes' set of potential identifiers on a single certificate. Our algorithm compares favorably to four others in trace-driven simulations. We have implemented our algorithm and found that it improved aggregate throughput by 20% in a widely heterogeneous system in our experiments.**

I. Introduction

Decentralized structured overlays and distributed hash tables proffer a unique vision of computing: each machine seamlessly contributes to and benefits from a large service-oriented network. This vision has yet to be realized, in part, because machines are not identical, because the workload applied to the system may be heavy-tailed, and because node availability and churn rates may change over time. Learning to adapt to these characteristics through load balancing in a decentralized, scalable, and secure manner is a step toward realizing this ideal of computing.

Several existing proposals for load balancing algorithms in this context have focused on ideal conditions [1], [26], [30], [32]. They have made unrealistic assumptions about node heterogeneity, workload skew, and node churn. In general, they have assumed that nodes are uniform, that there is no skew in the workload, and that nodes are neither arriving nor departing. Deployed systems do not adhere to these idealistic conditions [39], [45].

Other proposals have attempted to handle skew, churn, and heterogeneity [12], [20], [35]. Those that achieve good performance let nodes join as normal and then reactively position nodes to arbitrary locations in the namespace. Arbitrarily choosing identifiers (IDs) forfeits an important security goal for p2p systems: a verifiable identifier. Without verifiable IDs tying virtual overlay addresses to specific agents, application building blocks such as reputation [13], micropayments [46], and auctions [23] are not possible outside of a trusted network.

In this paper, we propose *k-Choices*, a load balancing algorithm for structured overlays that supports wide variation in skew, heterogeneity, and churn while retaining the security and application advantages afforded by verifiable IDs. At a high level, the algorithm works as follows: (a) each node generates a set of verifiable IDs based on a single unit of certified information; (b) at join time, a node greedily reduces discrepancies between capacity and load both for itself and for nodes that will be affected by its join; and (c) optionally, each node experiencing overload or underload may periodically probe the network and reposition itself to another element from its set of verifiable IDs. Minimizing discrepancies between load and capacity achieves load balance, and limiting IDs to a well-defined set keeps the algorithm secure.

This paper proceeds as follows. In Section II, we introduce our model and assumptions. In Section III, we present the *k-Choices* algorithm in detail. In Section IV, we review four state-of-the-art algorithms for load balancing in p2p systems. In Sections V and VI, we present results from trace-driven simulations where we vary system characteristics, including node heterogeneity, skew, and churn. We also present results from an implementation of *k-Choices*. Sections VII and VIII present related work and conclusions, respectively.

II. Model

In this section, we introduce our model and assumptions for load balancing in p2p systems.

Overload. Physical nodes, *i.e.*, computers, participate in p2p systems. Each node n_i has a capacity C_i, which corresponds to the maximum amount of load that node can process per unit time. Nodes create virtual servers (VSs), which join the p2p network. A node n might have j VSs $v_1, v_2, \ldots, v_j$, each with loads $w_1, w_2, \ldots, w_j$, respectively. Load is applied to nodes via their virtual servers. In a unit of time, node n_i might have load (work) $W_i = w_1 + w_2 + \ldots + w_j$.

Overload occurs when, for a node n_i, $W_i > C_i$. An overloaded node is not able to store objects given to it, route packets, or perform computation, depending on the application. A node fails to process requests that impose work beyond its capacity. Per unit time, the successful work per node is:

$$S_i = \begin{cases} W_i, & \text{if } (W_i \leq C_i) \\ C_i, & \text{otherwise} \end{cases}$$

The utilization of a node's n_i is W_i/C_i. Nodes may want to operate below their capacity C to prevent fluctuations in work-

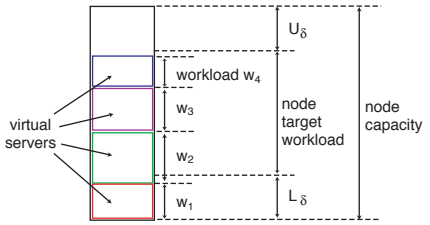

Fig. 1. Target and Capacity Workload.

load from temporarily overloading them. Using terminology from Rao *et al.* [35], we say a node n_i has an upper target U_i and slack U_δ such that $U_i = C_i - U_\delta$. If a node finds itself receiving more work than U_i, it considers itself overloaded. Nodes also have a lower target L_i below which they consider themselves underloaded. How a node responds to either of these conditions depends on the algorithm. An illustration of how we represent nodes is shown in Figure 1. We assume each node knows its capacity C and its upper and lower targets.

Each node stores its virtual servers in a set, called `VSset` of size `VSset.size`. Depending on the algorithm, this size may have an upper bound of `VSset.maxsize`.

Routing. Structured overlays allow routing of messages to destinations on top of an underlying network constantly undergoing topology change [36], [38], [43], [47]. Each message's destination ID is a number on the overlay's namespace D, *e.g.*, $D = 2^{160}$. Messages traverse overlay hops from a source VS to a destination VS. The number of hops is typically $O(log(N))$, where N is the current number of VSs.

Each VS has a unique ID chosen from the namespace D. In our model, the destination of a message is the VS with the next largest logical identifier on the namespace mod D. The VS with the next largest (smallest) ID is called the *successor* (*predecessor*). We denote the distance in the namespace between two virtual servers i and j with $dist(i, j)$.

Each structured overlay allows new VSs to join the system. In general, each VS join and departure requires $O(log(N))$ maintenance messages. Reactive load balancing algorithms use artificial join and departure to change IDs.

Network as Bottleneck. We focus on how load balancing algorithms function at the routing level. Blake and Rodrigues provide evidence that even in remote storage applications, network bandwidth is likely to be the primary bottleneck [2]. As storage becomes cheaper and cheaper relative to bandwidth, particularly "last-mile" bandwidth, this case will likely become more common. In compute-dominated scenarios, whether the processing or the network will be the bottleneck depends on the application. We let a node n_i's capacity C_i be the number of routing hops it can provide per unit time. We compare algorithms on the percentage of messages that successfully reach their destinations.

Security. A key issue in the operation of a p2p network is whether or not one assumes it may contain malicious nodes. A malicious node can subvert content or attempt to control particular portions of the identifier space. Attacks that center around the falsification of a node's identifier are called Sybil attacks [14]. Douceur outlines the main difficulties in allowing nodes to choose their own IDs. He shows that validating nodes must verify all other nodes' credentials simultaneously, an act that may exceed the verifier's resources.

A system may acquire a low level of security by requiring that IDs be based on the hash of the node's IP address [12]. However, falsifying IP addresses is straightforward; basing any level of authentication on IP addresses would not repel a determined attacker. For this reason, Castro *et al.* propose that each ID k is certified by a central authority, which generates k_{cert} [9]. This option is scalable because each node contacts this authority once, the first time it joins the system.

Instead of having this authority certify IDs, we propose that it certify a unique number x for each node, creating x_{cert}. Each node can then use this number to generate its own IDs using an ID-generating hash function h. For a node with ID k, a verifier verifies that $k = h(x_{cert})$ instead of $k = k_{cert}$. *k-Choices* creates a set of verifiable IDs by generating each $k = h(x_{cert} + c)$ where c has a well-known bound. We refer to x_{cert} as x below for purposes of presentation.

The *k-Choices* solution we propose retains this Sybil attack resilience. Algorithms that permit a node to relocate its virtual server to an arbitrary node ID location do not have this quality. Algorithms that do not allow for certified IDs can only be expected to function in a trusted environment.

System Characteristics. Although structured overlays are targeted to provide the framework for applications such as application-level multicast [8], distributed storage [10], [16], and publish-subscribe content distribution [34], [42], there are no benchmark workloads. Gummadi *et al.* and others have found Zipf query distributions in their trace analysis of Kazaa [3], [24], [39] and this distribution is common to many other usages (*e.g.*, web page file access [18], file popularity [17]). We examine load balancing under uniformly random and Zipf queries. A Zipf workload with parameter α means that destinations are ranked by popularity. Destination with rank i is α times more likely to be accessed than that with rank $i+1$.

A characteristic related to *skew* is *workload shift*. Shift refers to a change in workload skew. For example, on one day, one stored object might be the most popular, on the next, a different one might be, but the general distribution would be the same. Studies of object popularities in deployed p2p systems have found the existence of shifting Zipf skewed workloads [24].

A third characteristic is the distribution of node *capacities*. As is generally the case in p2p scenarios, bandwidth is the main capacity limiter [2]. In the traces which we draw from, node capacities vary by six orders-of-magnitude [39] and a simple function does not capture the trace bandwidth distribution well.

A final characteristic is the distribution of node joins and departures (*churn*). As we discuss in Section V, this cannot be captured with a simple rate λ. Instead, churn tends to be Pareto: heavy-tailed and memory-full. Nodes that have been in the system for a long time tend to remain longer than average [3]. Pareto distributions have two parameters, shape

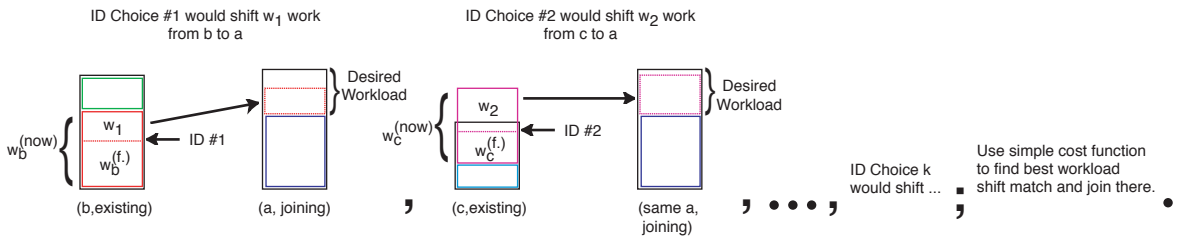

Fig. 2. As part of the *Join* process, *k-Choices* shifts workload for each of the VSs that are created.

K-CHOICES VS JOIN(t_a)

1 $K \leftarrow \{k_0 \leftarrow h(x+0), \ldots, k_{\kappa-1} \leftarrow h(x+\kappa-1)\}$
2 Remove in-use IDs from K
3 **for each** k **in** K
4 **do** Query $succ(k)$ for $w_s^{(n)}$ and t_s
5 $r \leftarrow \frac{dist(pred(k),k)}{dist(pred(k),succ(k))}$
6 $w_a \leftarrow r \times w_s^{(n)}$
7 $w_s^{(f)} \leftarrow (1-r) \times w_s^{(n)}$
8 $c \leftarrow |t_s - w_s^{(f)}| + |t_a - w_a| - |t_s - w_s^{(n)}|$
9 Join at k with minimum c
10 **return** w_a

K-CHOICES NODE JOIN(T)

1 $T \leftarrow (U_i + L_i)/2$
2 $i \leftarrow \kappa/2$
3 **while** $T > 0$ and $i > 0$
4 **do** $T \leftarrow T -$ K-CHOICES VS JOIN(T)
5 $i \leftarrow i - 1$

Fig. 3. *k-Choices* join algorithm. $w_s^{(n)}$ and $w_s^{(f)}$ denote the successor's work now and in the future, respectively.

α and scale β, and have a mean of $\frac{\alpha \times \beta}{\alpha - 1}$.

III. *k-Choices* ALGORITHM

k-Choices is a greedy, cost-based load balancing algorithm for structured overlays. It matches nodes' workload goals with guesses about how their choices of identifiers will affect both their own workloads and those of their neighbors. At each VS insertion, *k-Choices* minimizes the discrepancy between work and capacity by sampling from a small set of potential IDs. By limiting the number of potential IDs, *k-Choices* is practical for networks containing malicious participants.

k-Choices functions primarily at node join time as shown in `k-Choices Node Join` in Figure 3. When a node joins, it chooses a total target workload and an upper bound on the number of VSs to create (lines 1-2). Then, it invokes `k-Choices VS Join` and reduces its remaining capacity by the anticipated work of that VS (line 3). This continues until it has created $\kappa/2$ VSs or reached its target workload. Making several VSs together at join time amortizes the cost of sampling.

The join for each VS is composed of four steps, as shown in Figure 2 and in `k-Choices VS Join` in Figure 3. A small menu of potential IDs is chosen, limited by a well-known constant κ (lines 1-2). These IDs are verifiable because they are all based on the certified x and because κ is bounded. To verify that a node is using a valid ID, k, another node simply has to check that there exists some $i < \kappa$ such that $k = h(x+i)$. Next, each potential ID's successor is probed to discover what is likely to happen were this VS to be placed at this location (line 4). It guesses that the current work for this location will be split based on the percentage of the address space the joining VS will take on (lines 6-7). The node uses this to compute the change from the current situation (line 8). Each term in the cost function is the difference between target work and real work. The first two terms are the sum of the differences if this VS is created and the last is the current situation. We normalize each term based on the node's capacity. Thus, the lower the cost, the smaller the difference between target and actual work. The last step of the join process is to join at the ID with lowest cost. Because nodes set their targets lower than their capacities, if all nodes minimized the mismatch $m = |t - w| = 0$, then loss would be zero.

If nodes do not attempt to perform any additional load balancing after joining, we say they are *passive*. Being passive has its advantages: no additional churn is induced through VS relocation. However, over time one of the other potential IDs for this VS can become significantly better in terms of improving target/workload mismatch.

If we permit reselection of IDs, we say that *k-Choices* is *active*. To minimize network probing, nodes reselect only a single VS ID at a time. They pick the $v \in$ `VSset` with the maximum mismatch. They check if any new ID for v improves the aggregate mismatches of themselves and their neighbors by ϵ, a parameter that dampens improvements of minimal benefit. If it does, the movement is performed. ϵ is application-dependent: when a system is used for routing, moving will be relatively painless, as VSs can gracefully notify incoming pointers of their departure; if objects are stored and need to be sent over the network, the cost might be significantly greater. Nodes only examine the possibility of relocating if they are overloaded or underloaded. If nodes have relocated more than `VSset.size` times and are still overloaded or underloaded, they create or destroy VSs within the range $(1, \kappa)$. In practice we found that nodes did not create more than a handful of additional VSs.

k-Choices possesses several attractive features and makes certain assumptions. When run in *passive* mode, it adds no

reactive churn. In fact, without an active component, it *requires* natural churn. By making a good choice before routing is set up, objects are stored, or computations are started, *k-Choices* lessens or eliminates this reactive load balancing penalty. We assume that nodes do not lie or that Distributed Algorithmic Mechanism Design techniques could be used to encourage the truthfulness of the information they provide about load [19], [41], another reason why verifiable IDs are important. We also assume that the system is not primarily being used for range queries. Limited ID assignment provably cannot balance load in this case [26].

Note that VSs could keep more accurate track of where work is landing in their namespace to make w_a and $w_s^{(f)}$ more accurate. Instead, we decided to use a simple exponentially-weighted moving average to reduce the amount of state sent during probing.

Optimal ID Choice. *k-Choices* exhibits diminishing returns as κ approaches the size of the namespace D. When $\kappa = D$, each joining VS would sample every possible ID (assuming a perfect hash function). In fact, it is feasible to find the ID (or IDs) with the lowest cost by examining only a few variables for each existing VS. While even this sampling would be prohibitively expensive in an implementation, performing it "offline" within a simulator is not.

For each potential successor s, we know its target t_s and its actual work $w_s^{(n)}$. The goal is to find the percent of the address space r between $pred(s)$ and s that gives the minimum cost c and to find what the cost is for this s. The optimal ID choice will be the $pred(s) + r \times dist(pred(s), s)$ with the globally lowest cost. We know that $w_s^{(n)} \geq 0$ and that $m_s = |t_s - w_s^{(n)}|$ is fixed regardless of the r chosen. If we do not normalize for each node's capacity, there are four mutually exclusive cases for r and c:

case $t_s \leq 0$ and $t_a \geq w_s^{(n)}$:
$r = 1$; $c' = t_a - t_s + w_s^{(n)} - 2rw_s^{(n)}$

case $t_a \leq 0$ and $t_s \geq w_s^{(n)}$:
$r = 0$; $c' = t_s - t_a - w_s^{(n)} + 2rw_s^{(n)}$

case $t_a + t_s < w_s^{(n)}$:
$r \in (\frac{t_a}{w_s^{(n)}}, 1 - \frac{t_s}{w_s^{(n)}})$; $c' = w_s^{(n)} - t_a - t_s$

case $t_a + t_s \geq w_s^{(n)}$:
$r \in (1 - \frac{t_s}{w_s^{(n)}}, \frac{t_a}{w_s^{(n)}})$; $c' = t_a + t_s - w_s^{(n)}$

$c = c' - m_s$

In cases 1 and 2, we do not eliminate r because IDs cannot be identical. The actual choice will need to be a small distance away.

IV. Prior Load Balancing Techniques

In this section, we discuss the four existing load balancing algorithms against which we will compare *k-Choices*: *log(N) VS*, *Proportion*, *Transfer*, and *Threshold*. The first, *log(N) VS*, solely attempts to evenly partition the namespace between nodes, ignoring heterogeneity and skew. *Proportion* creates VSs in proportion to capacity at join time and makes adjustments based on workload. *Transfer* and *Threshold* use arbitrary VS relocation to adjust to skew and heterogeneity.

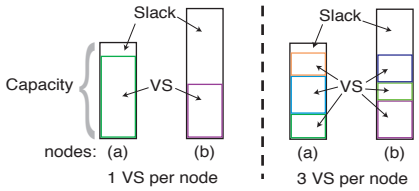

Fig. 4. *log(N) VS*: During join, a node can divide itself into several virtual servers, which then join independently. When all nodes do this, discrepancies in the average total namespace per node diminish.

Transfer and *Threshold*, in particular, are representative of the current state-of-the-art in load balancing algorithms for structured overlays. *Proportional* is particularly interesting because of its complete decentralization. *log(N) VS* allows us to show the pros and cons of pure namespace balancing. Because *Proportional* limits `VSset.size` to some well-known maximum, it also does not fundamentally change the security characteristics of the system. However, its performance is significantly inferior to *Transfer*, *Threshold*, and *k-Choices*.

A. $log(N)$ Virtual Servers

The simplest load balancing technique we discuss is *log(N) VS*. It balances node namespaces and does not permit arbitrary IDs. It was first introduced by Karger *et al.* [25].

The *log(N) VS* load balancing algorithm follows from the observation that randomly chosen node IDs do not uniformly cover the identifier space. In fact, the distribution of namespaces is roughly Poisson, with the largest being $O(log(N))$ times the average.

log(N) VS is predicated on the assumptions that workload and capacity are uniform. When these assumptions hold, if each node has a single VS, those few nodes at the tail become bottlenecks. By the Central Limit Theorem, the more VSs each node makes, the more normal (and balanced) the average (total) namespace of each node becomes. Because there are drawbacks to having too many VSs, this algorithm suggests that each node having $log(N)$ VS reaches a good compromise. All nodes then have average load within a constant factor. An illustration of this is shown in Figure 4. This technique is non-reactive: it makes no attempt to rebalance load after a node joins.

This algorithm works well for the case when its assumptions on capacities and workload distribution hold. However, increasing the number of VSs causes a few problems. First, it increases churn because when one node departs, it must take its $log(N)$ VSs with it, causing $log(N)$ times more adjustments to be made. Second, each node must hold $log(N)$ times as much routing state. Finally, because there are more VSs in the system, the number of hops per lookup (and latencies) increases. Proposals have been made to mitigate the last two problems, but they have not been evaluated [12].

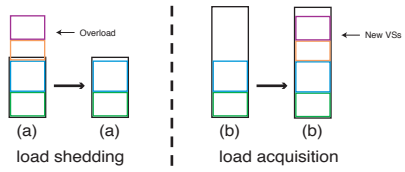

Fig. 5. *Proportion*: Underloaded nodes create new virtual servers (up to some maximum). Overloaded nodes destroy their own virtual servers (keeping one).

PROPORTION-ADJUST()
1 (Initially create VSs in proportion to capacity)
2 **if** overloaded and VSset.size > 1
3 **then** Delete VS that will best unload us
4 **if** underloaded and $W + \frac{W}{VSset.size} < U$
5 and VSset.size $<$ VSset.maxsize
6 **then** Create VS.id $\leftarrow h(x +$ VSset.size)

Fig. 6. *Proportion*'s Adjust algorithm.

Because several improvements to this basic namespace balancing concept have been proposed (see Section VII), the *log(N) VS* algorithm provides a baseline to suggest how this type of algorithm can be expected to perform under conditions of heterogeneity and skew in particular.

B. Proportion

Proportion targets heterogeneity primarily, not workload skew. An administrator initially configures a node with a number of VSs in proportion to its capacity. In addition, previously observed workload may be taken into account. After this initial step, each node adds or sheds load without any communication with other nodes. It was first proposed by Dabek *et al.* [12].

After setup, each node periodically follows `Proportion-Adjust`, shown in Figures 5 and 6. A node running *Proportion* independently creates and destroys virtual servers. If a node is overloaded and is running more than one virtual server, it selects the least loaded VS that will make it underloaded and deletes it (lines 2-3). If a node is underloaded and believes that adding a VS will not put it over its target load, it creates a virtual server (lines 4-6). Without any extra communication, underloaded nodes actively take on more work. The goal of the algorithm is that, over time, this will ease the burden on overloaded nodes because they will assume a smaller percentage of the workload as the number of VSs increases.

Load balancing in complete isolation has its drawbacks. First, a node with only a few VSs may not be able to form a good estimate of what the cost of creating a new one will be. Second, a meager machine still might be overloaded even if it is only running one VS. If a new physical server enters and has significantly less capacity than the current low-end servers, the system may take a long time to adjust to this new lowest common denominator. Third, if an overloaded node deletes one of its VSs, this may overload its neighbor, resulting in cascades

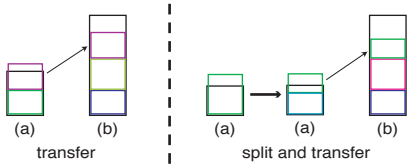

Fig. 7. *Transfer*: Overloaded nodes attempt to transfer virtual servers to underloaded nodes. If they only have one VS and are still overloaded, they split the VS in two equal halves (and transfer one).

TRANSFER()
1 **if** !overloaded
2 **then return**
3 **if** VSset.size > 1
4 **then** Contact node n at random
5 Choose $v \in$ VSset such that:
6 (a) Transferring v to n will not overload n
7 (b) v is the least loaded virtual server
8 that will halt overload;
9 Failing that, let v be most loaded VS
10 **else** $v \in$ VSset [0]
11 Create VS.id $\leftarrow v.id + \frac{dist(pred(v),v)}{2}$ mod D
12 TRANSFER

Fig. 8. *Transfer*'s Split and Transfer algorithm.

of deletes. Finally, when the system is underloaded, *Proportion* can cause all nodes to create their maximum number of VSs, greatly increasing state, routing hops, and churn.

C. Transfer

Transfer focuses on actively unloading overloaded nodes. Instead of having underloaded nodes take on more work in isolation like *Proportion*, overloaded nodes following the *Transfer* algorithm actively seek out underloaded nodes to inquire about load transfer. Thus, nodes select arbitrary IDs at two points: when they split and when they receive transfers. This idea was first proposed by Rao *et al.* [35].

The algorithm works as shown in Figures 7 and 8. If a node is overloaded and it has only one VS, then it splits the VS into two equal parts (line 11). If a node is overloaded and if it has more than one VS (one of which may have just been created via a split), it attempts to contact an underloaded node and transfer an appropriate VS (lines 3-9). The transfer fails if all VSs would overload the potential receiver.

Transfer moves work around effectively. Nodes are never transferred work they cannot handle. However, when the system is near capacity, overloaded nodes may need to contact many others to perform a successful transfer.

Transfer has a few permutations. The scheme presented here and used in the experiments is known as "one-to-one" because one node contacts a single other node per unit time. The same work also proposed "one-to-many" and "many-to-many" variations and found they utilized nodes similarly. Godfrey *et al.* propose a more complex variation where nodes randomly

```
THRESHOLD(v, t)
 1  v.level_t ← ⌊log_ρ(v.util / c)⌋
 2  if v.level_t ≤ v.level_{t−1}
 3     then return
 4  v' ← adjacent neighbor with lowest level
 5  if v'.level_t < v.level_t
 6     then Δ ← (1 − 1/ρ) × dist(pred(v), v)
 7          if v' = pred(v)
 8             then pred(v).id ← pred(v).id + Δ
 9             else v.id ← v.id − Δ
10     else  /* find new predecessor */
11          S ← set of log(N) random VSs
12          Choose s ∈ S such that:
13          (a) s is the least utilized
14          (b) w_s + w_succ(s) < U_succ(s)
15          s.id ← pred(v).id + dist(pred(v),v)/2 mod D
```

Fig. 9. *Threshold*'s load balancing algorithm.

choose one of a handful of well-known exchange points that periodically reallocate work [22].

D. Threshold

Threshold focuses on keeping all nodes' utilizations within a ratio ρ, as opposed to between target overload and underload values like the other algorithms. It also keeps the number of VSs to a minimum (one per node). *Threshold* allows the selection of arbitrary IDs in both its neighbor adjustment and VS relocation phases. We present a modified version of Ganesan *et al.*'s algorithm [20]. We made two modifications: (a) we use utilization instead of workload because the original algorithm assumes homogeneous capacities and (b) nodes only initiate rebalancing when they increase in level.

Each node has exactly one VS whose ID is initially chosen at random. The rebalancing algorithm shown in Figure 9 is called by a node with VS v at time t. Nodes set their current utilization level such that a level increases by one if work has increased by a factor ρ, where c is some small constant (line 1). If a node's level has increased, it starts load balancing (line 2). It first attempts to make adjustments with its neighbors (lines 4-9). VS v first sees if local adjustments in the IDs of its successor or predecessor are feasible, potentially shifting some work to them. If the predecessor is lightly loaded compared to v, its ID is shifted toward v (line 8). This action should result in its taking some of v's load. v can also move its own ID closer to its predecessor, which potentially shifts work from v to its successor (line 9). If making neighbor adjustments fails, it relocates a lightly loaded node to be its new predecessor (lines 10-15). Ties between successor and predecessor are broken arbitrarily. If neither of these options is available, v attempts to find a new predecessor to take (ideally) half of v's load. v picks a set of VS's at random and relocates the most underutilized whose departure will not overload its successor (line 11-15).

Threshold diminishes the importance of the tuning parameter δ, but introduces a significant parameter in ρ. If ρ is too large, load balancing will occur too slowly. If it is too small, nodes will make many unnecessary adjustments. A compromise is to set ρ for slow adjustments but to induce load balancing if the node becomes overloaded even if levels have not changed. We included this compromise in our implementation. Because *Threshold* always chooses the least utilized VS to relocate, VSs with very high capacity (and therefore low utilization) may tend to be relocated frequently.

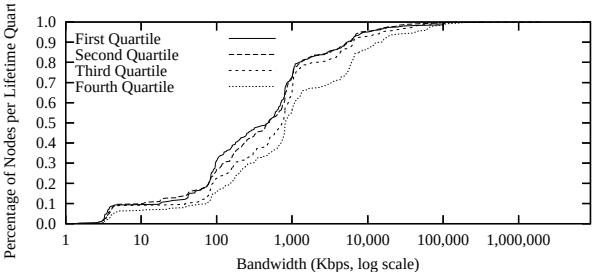

Fig. 10. CDFs of Downstream Bandwidth per Average Lifetime Quartile.

V. SIMULATOR

We built a simulator to compare the load balancing algorithms discussed in Sections III and IV. While simulators exist for several p2p algorithms, none supports virtual servers or drops packets under overload [6], [21]. This section describes the simulator and how queries succeed and fail.

The simulator operates in discrete time steps. Each time step consists of the following phases: node arrival and departure, routing table updates, queries, and load balancing.

Node arrival and departure. At each step, nodes arrive and depart. A typical method for generating birth/death processes is to assume Poisson distributed lifetimes (and deathtimes) with some mean λ [28], [29], [33]. However, Bustamente *et al.* have found, through trace analysis of Gnutella, that p2p systems do not follow this memory-less distribution and, in fact, approximate longer-tailed Pareto distributions more closely [3].

For our trace-based experiment, we use a Gnutella trace directly [40]. Because we wanted to include the correlation between node lifetimes and their capacities, we extracted from the trace the nodes for which upstream or downstream bandwidths were available. The extracted traces consist of 5508 nodes joining and leaving the Gnutella network for 60 hours. We based churn on the times when the IP addresses of the node could be reached in the trace. The median lifetime of a node was about one hour. We converted from the trace's bandwidth information to messages per second by assuming an average message size of 10KB. The median node could forward 191 messages per second. We show the bandwidth distribution and modest correlation between bandwidth and lifetime in Figure 10. The trace does not include any topology information, and we do not include any in our simulation.

For the experiment where we vary node lifetime, we instead generated several Pareto birth/death distributions with varying

mean. Because Pareto distributions can take a long time to stabilize, we only took a snapshot of the distribution after this stabilization had occurred. We used $\alpha = 2$ and varied β, avoiding instabilities with smaller values of α [11].

One unnatural aspect of both the synthetic and trace-driven churn is a large number of births at the beginning of each experiment. Because each algorithm needs some workload information to operate, they did not activate until a short period into each experiment. We choose an activation time of 400 seconds, as this was when all of the nodes in the Gnutella trace had first joined. In addition, we recorded statistics only for the second half of each experiment.

Routing Table Updates. New VSs start off with an empty routing table. They follow the Chord mechanism to find a node to fill each of their $log(N)$ slots [43]. Each node with ID a fills its i^{th} entry with the node whose ID is the successor to $a + 2^i \bmod D$.

Each routing table entry, or *finger*, has a timeout set to 30 seconds on average. Each time this finger is used successfully, the timeout is reset. This simple technique typically has been found to be effective in supressing maintenance messages [7]. Nodes do not invalidate their fingers on a failed attempt at forwarding because they do not know if the receiver is dead or overloaded. When nodes gracefully change their VSs' identifiers, other virtual servers pointing to them are notified. When nodes die, VSs pointing to them are not notified (*i.e.,* death is ungraceful), as would be the case were a user to switch off his or her machine. Nodes make certain their successor fingers are always valid.

Queries. Queries initiate from nodes uniformly at random with destinations chosen from either uniform or Zipf distributions, depending on the experiment. Each hop in the query uses the appropriate finger to route toward the destination. Each use of a VS for routing or maintenance adds one unit of load per that VS's node. If a hop is to a node whose load for that unit of time matches or exceeds its capacity, the query fails. Queries succeed when they reach their destination.

Load Balancing. Nodes check on their load balance once every 30 seconds on average. They determine their utilization by examining an exponentially-weighted moving average of the work their VSs perform. They check if they are above or below their targets, which were set to $.95\times$ and $.05\times$ capacity, respectively. If they are out-of-balance, they perform whichever reactive algorithm is currently under test.

`VSset.maxsize` was set to 128 as suggested by the Chord research group. Each node running *Transfer* began with five VSs as suggested by Rao *et al.* [35].

VI. Results

The following summarizes our experimental results:

- In Section VI-A, we show that simple namespace balancing is effective when workloads are uniform and node capacities are a constant (its assumed conditions). We portray the diminished value in this form of balancing as workload becomes skewed.

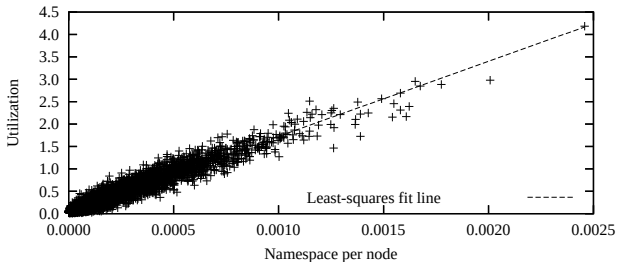

Fig. 11. There exists a strong correlation between a node's namespace and its utilization when the workload is uniform and capacities are constant.

TABLE I
Workload Skew under Namespace Balancing

Skew	Random		Balanced	
	r^2	Succ. %	r^2	Succ. %
Uniform	0.95	0.59	> 0.99	1.00
Zipf ($\alpha = 0.8$)	0.83	0.46	0.83	0.53
Zipf ($\alpha = 1.2$)	0.61	0.27	0.57	0.29
Zipf ($\alpha = 2.4$)	0.36	0.03	0.31	0.04

- In Section VI-B, we explore parameter choices for κ for *k-Choices* and find that $\kappa = 8$ performs well for the workloads we examine.
- In Section VI-C, we compare how the algorithms respond to varying applied workload when nodes follow trace-based churn and capacity. We find that only *k-Choices* and *Transfer* can support large amounts of skewed load.
- We show that *k-Choices* can support high churn rates in Section VI-D.
- Section VI-E portrays that *k-Choices* sustains high success rates throughout shifting workloads. We also find that *Transfer*, *Threshold*, *Proportion* exhibit inconsistent results over time.
- In Section VI-F, we show that none of the algorithms can support very skewed workloads (*e.g.,* $\alpha = 4.8$) and that they increase in variance as skew increases.
- In Section VI-G, we find our implementation of *k-Choices* within Pastry [38] improves throughput by 20% on an implementation in a heterogeneous-bandwidth networked environment.

More information on the experiments, simulator, tuning, and validation is available in the accompanying technical report [27].

A. Namespace Balancing

These first experiments confirm that, under conditions of constant or near-constant capacity and uniform query distribution, simple namespace balancing is highly effective. However, when either of these conditions fails to hold, it is not.

In order to see the correlation between a node's namespace and its utilization, we ran a simple set of experiments in which we varied workload skew in a system that was performing no load balancing. We monitored the incoming routing and maintenance messages for each node and compared this to

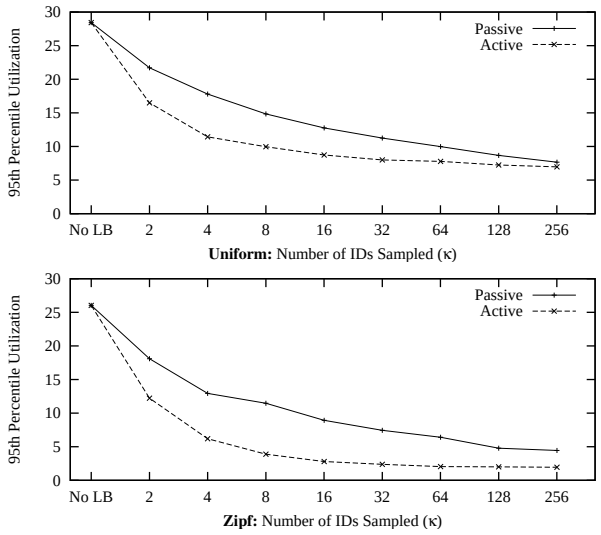

Fig. 12. *k-Choices* 95^{th} percentile utilization decreases as κ increases.

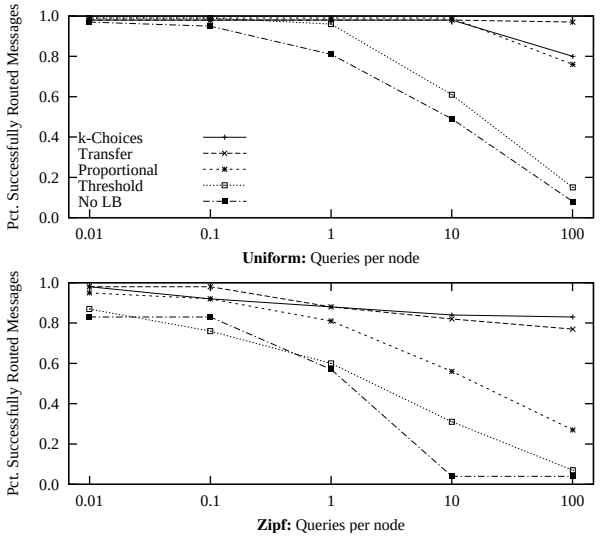

Fig. 13. Percent of successfully routed queries for trace-driven simulations with varying loads.

the fraction of the ID space for which that node was used as a hop or destination. We ran two sets of experiments: one where VS identifiers were chosen at random and a second where they were set offline to be exactly equal. This second case shows the best that namespace balancing could achieve. We used 4096 nodes and set all node capacities so that they could route 100 messages per second. No churn was used because the exactly equal ID computation is only performed offline. We varied workload skew from uniform to Zipf with $\alpha = 2.4$. Because no active algorithm was used and there was no churn, each experiment stabilized immediately. Every node had one virtual server and there were 40960 queries per second (10 queries per alive node).

We plot the correlation between namespace and node utilization for a uniform workload in Figure 11. As is expected, the average namespace per node is $\frac{1}{4096} \approx .0002$. Because no load balancing is used, the distribution of namespaces is long-tailed. Analytically, the largest distance between two VSs should be $\frac{1}{4096} \times log(4096) \approx .0029$, close to the measured value of $.0025$. Utilizations with random IDs ranged from almost 0 to about 4. In contrast, the case where the namespaces were completely balanced yielded an extremely small range of utilizations from 0.55 to 0.57.

As we relax the assumption that workloads are uniform, the benefit in perfectly uniform address spaces declines. Table I shows how the correlation and success rates for queries decline as workload skew increases. Separate experiments confirm a similar decline as heterogeneity in nodes' capacities changes from a constant. *We can conclude from this that, in order to achieve reasonable performance, a load balancing algorithm must include some workload parameter and cannot aim for address space balancing alone.*

B. Varying κ

The second set of experiments explores *k-Choices* parameters for Gnutella-like systems. Our goal was to find a reasonable set of parameters for the subsequent experiments.

We generated a synthetic churn trace of 4k nodes with Pareto distributed average lifetimes of 60 minutes and a Gnutella-like capacity distribution with average capacity of 100 messages/second. Each node initiated 10 queries/second. We ran each experiment for three hours and monitored node utilization. We varied κ and ran *k-Choices* in *active* and *passive* modes.

The 95^{th} percentile utilizations are plotted in Figure 12. When $\kappa = 1$, *k-Choices* is not in use, showing the situation without any load balancing. The results show that *active k-Choices* lowers utilization at a significantly faster rate than *passive* does as κ increases. In both lookup scenarios, the 95^{th} percentile utilizations do not decrease much beyond when $\kappa = 8$ in *active* mode. The results also show that a skewed query distribution ($\alpha = 1.2$) has minimal impact on utilization for *k-Choices*. In fact, it even lowers peak utilization as nodes with more bandwidth are able to position their VSs where the workload is concentrated. As noted above, there are substantial drawbacks to large numbers of VSs per node and to setting κ to a large value (*e.g.,* large numbers of probes). Therefore, we used $\kappa = 8$ in subsequent experiments, unless otherwise noted. As these results portend, preliminary experiments with Optimal ID choice suggest that *k-Choices* works well without a huge sampling of IDs. We also experimented with values for ϵ, which we set to 0.25 in our experiments. These results show that *k-Choices* needs only a small number of choices to produce a substantial decrease in node utilization.

We ran similar experiments to find good parameters for *Threshold*. Its two parameters τ and c were set to 8 and 0.01 respectively.

C. Trace Results

Our third experiment examines how the load balancing algorithms responded to different degrees of applied workload

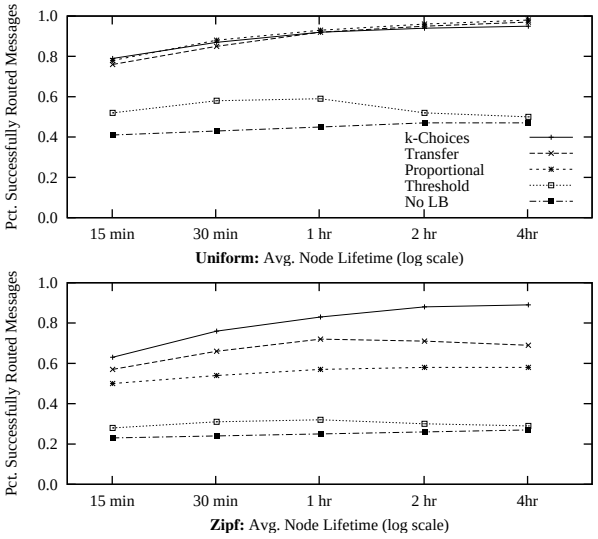

Fig. 14. Percentage of successfully routed queries for varying rates of churn.

using trace-driven churn and capacity. In almost all cases, we found *k-Choices* performed the same as or better than the other algorithms.

Each experiment used the Gnutella trace as described in Section V. Each ran for twelve hours with statistics recorded for the second half of the experiment. We varied the applied query load by orders-of-magnitude and recorded the percentage of queries that reached their destination. This experiment captures factors such as artificial churn and large numbers of VSs per node that some of the algorithms induce.

We plot the results in Figure 13. They show that all of the algorithms, except for *Threshold*, can sustain high success rates when queries are uniform, although *k-Choices* and *Transfer* do slightly better *Proportion*. At 100 queries/node, the 95^{th} percentile of the number of VSs/node was 128 for *Proportion* (the maximum), compared to 1.9 for *k-Choices* and 16.1 for *Transfer*. Performance for *k-Choices* in *passive* mode declines after 10 queries/node. We plot $\kappa = 16$; $\kappa = 8$ performed about 10% worse and $\kappa = 64$ performed about 10% better at this workload level.

When queries are skewed ($\alpha = 1.2$), only *k-Choices* and *Transfer* can sustain high query rates. At this level, the other algorithms are unable to maintain low utilization of low capacity nodes. *Log(N) VS* performed worse than *No Load Balancing* in these experiments and is not shown in the figures.

D. Varying Churn

Because *k-Choices* helps nodes make good load balancing choices proactively, we hypothesized that at high churn rates, it would offer better performance than the other algorithms. To test this, we created a set of synthetic churn traces with varying average lifetimes and used the same capacity distribution from the trace. We ran each algorithm with each node initiating 10 queries per second on average.

We plot the results from uniform and skewed ($\alpha = 1.2$) query distributions in Figure 14. The data confirm our hypothesis that *k-Choices* adapts well to rates of high churn. We found that both *Transfer* and *Proportion* were able to sustain high natural churn rates for uniform queries, but that they induced $1.1 - 1.5\times$ and $5 - 10\times$ more artificial churn, respectively, than *k-Choices*. Again, the variation in success rates is more prominent with skewed queries. This is because *k-Choices* monitors workload before insertion. *No Load Balancing* improves slightly as lifetimes increase because fingers remain valid for longer. Again *log(N) VS* had worse performance than *No Load Balancing*.

In both uniform and Zipf, *Threshold's* success rate declines as nodes' lifetimes increase. This occurs because *Threshold* makes the gaps between VSs so non-uniform that it significantly increases the average number of hops, *e.g.,* from 5.6 for 15 minute lifetimes to 7.3 for 4 hour lifetimes for uniform queries. Because queries are taking more hops and nodes are similarly load balanced, each query is less likely to succeed.

E. Workload Shift

For the fifth simulation experiment, we wanted to see how the algorithms responded to workload shifts. We ran each algorithm using trace-driven churn and capacity for ten hours. Halfway through each run, we changed the query destinations from one moderately skewed set to another (both with $\alpha = 1.2$). We recorded statistics throughout the trace. As noted, each algorithm activates after 400 seconds. Each node initiated 10 queries per second on average.

We monitored success rates and VS activity. VS activity captures the amount of state transfer that occurs due to natural and artificial churn. When a node enters or leaves the system, the number of VS actions equals the number of VSs in use. Creating or destroying a VS is also a VS action. Each *k-Choices* and *Threshold* `relocate` is two VS actions; each `transfer` is one. Conservatively, we did not include *Threshold*'s `neighbor-adjustments` or *Transfer*'s `splits` as VS actions.

The results are plotted in Figure 15. We show the success rate on the left y-axis and VS activity on the right y-axis. The results show that *active k-Choices* sustains $> 75\%$ success rates, recovering immediately after the workload shift. *Passive k-Choices* (not shown) gradually plateaus at about 40%. We found that in systems with higher rates of churn, *passive* reached equilibrium more quickly. As soon as *active k-Choices* is activated, the success rates dramatically increase. With current tuning, however, *active* produces an order-of-magnitude more VS activity than *passive*. After the shift, *k-Choices active* settles to a slightly lower success rate because queries heading to the new highest ranked spot take slightly more hops per average query: a change from 6.8 to 7.3.

Proportion, *Transfer*, and *Threshold* all portray greater variance in success rates than *k-Choices*. *Proportion* exhibits the greatest average VS activity and has the largest average hop count at 10.5 hops per successful query. The performance of *Threshold* steadily declines as its gaps become tightly

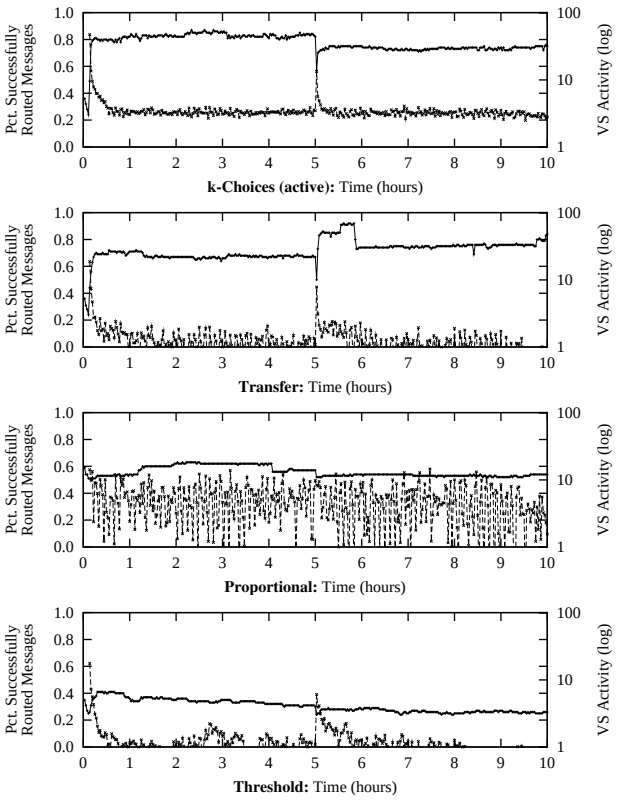

Fig. 15. Plot of success rate and VS activity during a workload shift.

clustered. That *Transfer* stabilizes at different levels had two causes. First, a burst of births soon after the shift caused more accurate fingers than average and a burst of deaths at 6 hours caused the decline because many fingers became invalid. Second, after the shift, the average path to highest ranked destination was fewer hops than before. Although to a lesser extent than *Threshold*, *Transfer*'s hop count steadily rises as nodes move to arbitrarily compressed locations.

F. Varying Skew

Some workloads are heavily skewed and several of the algorithms were able to support up to $\alpha = 1.2$. We wanted to examine how much skew they might support. To test this, we used the synthetic 60 minute average lifetime trace and the capacity distribution from the trace as we varied α. As before, we ran each algorithm with each node initiating 10 queries per second on average.

We plot the results in Figure 16. They show that none of the algorithms can support an extremely skewed workload, *e.g.*, one where the top destination is almost $5\times$ that of the next rank. Not only do the algorithms decline in their average success rates, but they also all become less stable. For example, the standard deviation of success rates sampled over time for *k-Choices* at *uniform* is 0.001% and at $\alpha = 4.8$ it is 8%. To see if increasing κ had an impact at high skew, we ran *k-Choices* with $\kappa = 16$. We found that it performed better (at 14%) than $\kappa = 8$, but also exhibited high variance.

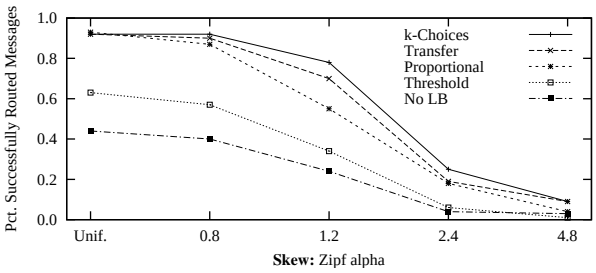

Fig. 16. Percentage of successfully routed queries for varying rates of skew.

TABLE II
SUCCESSFUL QUERIES FOR EMULAB EXPERIMENT

	Completed queries	
BW (MB/s)	*No LB*	*k-Choices*
.4	4341	5879 (+35%)
1	16672	20217 (+21%)
4	24025	29537 (+23%)
40	23331	26224 (+12%)
All	68370	81858 (+20%)

G. Emulab Experiment

To examine *k-Choices*'s effect on a working system, we implemented it within Pastry and ran a query-and-download scenario. Our primary goal was to measure changes in throughput with *k-Choices* using a fairly large real topology. Our use of nearest-neighbor-based Pastry demonstrates that *k-Choices* generalizes beyond Chord semantics. We based our *k-Choices* implementation on FreePastry [15]. We ran *k-Choices* in *passive* mode with $\kappa = 16$. We used 1 VS per node because FreePastry does not currently support multiple VSs. We were required to anticipate load based on namespace distances because low bandwidth nodes were unable to successfully join the network when queries were already taking place. For this same reason, queries were only for uniformly distributed destinations. If the destination responded, each node attempted to download an 8KB block. A query completed if both the query and download were successful.

We ran our experiment on Emulab, a testbed for networking research that supports precise bandwidth tuning [44]. The topology consisted of 256 nodes. There were 64 nodes of each bandwidth level; the levels were 40Mb/s, 4Mb/s, 1Mb/s, and 0.4Mb/s. Although Emulab has been working on making their system more scalable to support larger experiments, at the time, this was the largest topology we could run. Table II shows the total number of queries completed by bandwidth type. Each value is averaged over two trials that consisted of one hour of queries. All nodes used one of the 40MB/s nodes as their bootstrap. As a result, they were frequently in other node's routing tables and had a higher message routing workload. This is why their completed queries are fewer than the 4MB/s nodes. As expected, the average number of hops was a just less than 2, with minimal variance. The main experimental result, however, is that a 20% improvement in throughput confirms that *k-Choices* can have a substantial

positive impact on performance in a heterogeneous topology while retaining the important security properties of verifiable IDs.

VII. Related Work

Object load balancing. We have oriented our examination of load balancing around routing, where a node request must reach the destination ID for it to be successful. If the network is instead being used for storage, other load balancing techniques can be applied. Byers *et al.* describe a technique that hashes data to be stored using two distinct hash functions, providing two potential locations [4]. The less loaded of the two possibilities is chosen. During data lookup, the query must contact both possible storage locations, or appropriate forwarding pointers must be used. Under uniform workload and capacity assumptions and with no churn, they have recently generalized this result to show that the maximum load at any server is $loglog(N)/log(d) + O(1)$ where d is the number of choices [5]. Their method is an example of "the power of two choices" [31]. Our ID selection process is similar in spirit, in that we also use multiple hash functions, although here we do so to provide VSs with a menu of identifiers.

Objects may be cached in the network to reduce hot spots or overload. Roussopoulos and Baker develop a cooperative request scheme where nodes direct requests toward the highest capacity replica [37]. They assume that the source of each lookup is aware of the capacity of each possible replica holder. Sources of requests learn the replicas by first contacting the root of the query, a key's primary storage node, so it must still perform some work for their method to function.

These storage-oriented load balancing techniques are orthogonal and complementary to the methods examined in this paper, including *k-Choices*. For example, *k-Choices* reduces an overburdened node's namespace, preventing it from being contacted in the first place, and Roussopoulos' technique prevents it from being contacted frequently after the replica set is known.

Namespace balancing. While the simple $log(N)$ VSs per node achieves $O(1/N)$ namespace balance per node, more recent algorithms have achieved tighter bounds with fewer virtual servers. These algorithms are based on the assumptions that the capacity of nodes and workloads are uniform; they do not include any workload scaling parameter. Because of these factors, they would approximate the behavior and results of the $log(N)$ VS algorithm. If they did achieve perfect namespace balancing at zero cost, they could be expected to perform as Balanced does in Table I.

Four algorithms fall into this category. First, Karger and Ruhl propose that each node has $log(N)$ potential IDs, only one of which is activated at once [26]. Nodes activate and deactivate their VSs to balance the distance between themselves and their successor. Because this algorithm allocates nodes a limited number of IDs, it has stronger security properties than the remainder of this group. Second, Manku's algorithm reduces the ratio of the largest to the smallest partition to at most 4 w.h.p. and has low arrival and departure cost [30]. Third and fourth, Adler [1] and Naor [32] also have low cost algorithms to achieve namespace balancing based on unlimited virtual server movement. Both algorithms depend on the history of node IDs that each node has used and their analyses are given only for the insertions, not deletions, cases.

Range queries. While we have examined uniform and Zipf query distributions in our simulations, we have not examined load balancing algorithms targeted at p2p systems when performing range queries are common. However, if one considers using a p2p system more like a typical database where each node is analagous to a disk, it is clear that ordering data by key might be warranted. We are aware of two load balancing algorithms that are targeted for this new domain [20], [26]. We evaluated Ganesan's *Threshold* in this paper. Both require unlimited ID selection and, therefore, suffer from Sybil attack liabilities, making them unsuitable for non-cooperative environments. However, it is unlikely that a load balancing technique for range queries exists that supports scalable secure IDs.

VIII. Conclusions

We introduced a novel anticipatory load matching algorithm for balancing load in peer-to-peer networks. This algorithm makes explicit the workload assignment problem that previous work attempted to solve implicitly. The algorithm works preemptively as the node is joining to shift the "right" amount of work to the joining node. Optionally, it can continue to readjust workload mismatch over time.

After examining the *k-Choices* algorithm independently, we benchmarked its performance and that of other load balancing algorithms for structured overlays under conditions of node heterogeneity, skew, churn, and workload shift using trace-based simulations.

Prior work on load balancing for p2p systems has either focused on namespace balancing or on systems with more heterogeneous characteristics. We showed that even perfect namespace balancing results in poor performance under realistic conditions. Prior algorithms that do work well under these conditions, *Transfer* and *Threshold*, both allow the selection of arbitrary IDs, severely circumscribing their utility on insecure networks. We have shown that *k-Choices* can provide good load balancing under realistic conditions while retaining strong security properties necessary for wide-area applications.

Acknowledgment

The authors would like to thank Miguel Castro, Antony Rowstron, and Michael Mitzenmacher for helpful discussions.

References

[1] M. Adler, E. Halperin, R. Karp, and V. Vazirani. A Stochastic Process on the Hypercube with Applications to Peer-to-Peer Networks. In *STOC 2003*, San Diego, CA, June 2003.

[2] C. Blake and R. Rodrigues. High Availability, Scalable Storage, Dynamic Peer Networks: Pick Two. In *Proceedings of HotOS IX*, Lihue, HI, May 2003.

[3] F. Bustamante and Y. Qiao. Friendships that last: Peer lifespan and its role in P2P protocols. In *Eighth International Workshop on Web Content Caching and Distribution*, Hawthorne, NY, October 2003.

[4] J. Byers, J. Considine, and M. Mitzenmacher. Simple Load Balancing for Distributed Hash Tables. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, February 2003.

[5] J. Byers, J. Considine, and M. Mitzenmacher. Geometric Generalizations of the Power of Two Choices. In *SPAA 2004*, Barcelona, Spain, June 2004.

[6] M. Castro, M. Costa, A. Kermarrec, and A. Rowstron. SimPastry. `http://www.research.microsoft.com/~antr`.

[7] M. Castro, M. Costa, and A. Rowstron. Performance and Dependability of Structured Peer-to-Peer Overlays. In *Dependable Systems and Networks*, Florence, Italy, June 2004.

[8] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-Bandwidth Multicast in Cooperative Environments. In *Proceedings of the 19th ACM SOSP*, Bolton Landing, NY, October 2003.

[9] M. Castro, P. Drushel, A. Ganesh, A. Rowstron, and D. Wallach. Secure Routing for Structured Peer-to-Peer Overlay Networks. In *OSDI '02*, Boston, MA, 2002.

[10] L. Cox and B. Noble. Samsara: Honor Among Thieves in Peer-to-Peer Storage. In *Proceedings of the 19th ACM SOSP*, Bolton Landing, NY, October 2003.

[11] M. E. Crovella and L. Lipsky. Long-lasting transient conditions in simulations with heavy-tailed workloads. In *Proceedings of the 1997 Winter Simulation Conference*, December 1997.

[12] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, October 2001.

[13] R. Dingledine, N. Mathewson, and P. Syverson. Reputation in P2P Anonymity Systems. In *Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, June 2003.

[14] J. Douceur. The Sybil Attack. In *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, MA, March 2002.

[15] P. Druschel, R. Gil, Y. Hu, S. Iyer, A. Ladd, A. Mislove, A. Nandi, A. Post, C. Reis, A. Singh, and R. Zhang. Rice FreePastry implementation. `http://www.cs.rice.edu/CS/Systems/Pastry/FreePastry`.

[16] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, October 2001.

[17] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS tracing of Email Workloads. In *Proceedings of the 2003 USENIX Conference on File and Storage Technology*, San Francisco, CA, March 2003.

[18] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 2000.

[19] J. Feigenbaum and S. Shenker. Distributed Algorithmic Mechanism Design: Recent Results and Future Directions. In *Sixth International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, September 2002.

[20] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online Balancing of Range-Partitioned Data with Applications to Peer-to-Peer Systems. In *VLDB 2004*, Toronto, September 2004.

[21] T. Gil, F. Kaashoek, J. Li, R. Morris, and J. Stribling. p2psim. `http://www.pdos.lcs.mit.edu/p2psim/`.

[22] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load Balancing in Dynamic Structured P2P Systems. In *INFOCOM 2004*, Hong Kong, March 2004.

[23] B. Gross and A. Acquisti. Balances of Power on eBay: Peers or Unequals? In *Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, June 2003.

[24] K. Gummadi, R. Dunn, S. Saroiu, S. Gribble, H. Levy, and J. Zahorjan. Measurement, Modeling, and Analysis of a Peer-to-Peer File-Sharing Workload. In *Proceedings of the 19th ACM SOSP*, Bolton Landing, NY, October 2003.

[25] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *ACM Symposium on Theory of Computing*, El Paso, TX, May 1997.

[26] D. Karger and M. Ruhl. Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems. In *SPAA 2004*, Barcelona, Spain, June 2004.

[27] J. Ledlie and M. Seltzer. Distributed, secure load balancing with skew, heterogeneity, and churn. Technical Report TR-31-04, Harvard University, December 2004.

[28] J. Ledlie, J. Taylor, L. Serban, and M. Seltzer. Self-organization in peer-to-peer systems. In *Tenth SIGOPS European Workshop*, September 2002.

[29] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the Evolution of Peer-to-Peer Networks. In *PODC 2002*, July 2002.

[30] G. S. Manku. Balanced Binary Trees for ID Management and Load Balance in Distributed Hash Tables. In *PODC 2004*, St. John's, Newfoundland, Canada, July 2004.

[31] M. Mitzenmacher, A. Richa, and R. Sitaraman. *The Power of Two Choices: A Survey of Techniques and Results*. Kluwer Academic Publishers, Norwell, MA, 2001.

[32] M. Naor and U. Wieder. Novel Architectures for P2P Applications: the Continuous-Discrete Approach. In *Fifteenth ACM Symposium on Parallelism in Algorithms and Architectures*, San Diego, CA, June 2003.

[33] G. Pandurangan, P. Raghavan, and E. Upfal. Building low-diameter p2p networks. In *IEEE Symposium on Foundations of Computer Science*, pages 492–499, 2001.

[34] P. Pietzuch and S. Bhola. Congestion Control in a Reliable Scalable Message-Oriented Middleware. In *Middleware 2003*, Rio de Janeiro, Brazil, June 2003.

[35] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load Balancing in Structured P2P Systems. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, February 2003.

[36] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proceedings of the ACM SIGCOMM '01 Conference*, August 2001.

[37] M. Roussopoulos and M. Baker. Practical Load Balancing for Content Requests in Peer-to-Peer Networks. Research Report cs.NI/0209023, Stanford University, January 2003.

[38] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, November 2001.

[39] S. Saroiu, K. P. Gummadi, R. J. Dunn, S. D. Gribble, , and H. M. Levy. An analysis of internet content delivery systems. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, December 2002.

[40] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of the Multimedia Computing and Networking (MMCN)*, San Jose, CA, January 2002.

[41] J. Shneidman and D. Parkes. Specification Faithfulness in Networks with Rational Nodes. In *PODC 2004*, July 2004.

[42] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. In *SIGCOMM '02*, Pittsburg, PA, August 2002.

[43] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, August 2001.

[44] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, , and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, December 2002.

[45] B. Wilcox-O'Hearn. Experiences deploying a large-scale emergent network. In *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, MA, March 2002.

[46] B. Yan and H. Garcia-Molina. PPay: Micropayments for Peer to Peer Systems. In *ACM Conference on Computer and Communications Security*, Washington, D.C., October 2003.

[47] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Research Report UCB/CSD-01-1141, U.C. Berkeley, April 2001.